MONASH University

A/Prof Lindsay Kleeman/Dr David Boland
Clive Maynard

ECE3073 / TRC3300
Computer Systems

Deadlock in Real Time Systems

WARNING

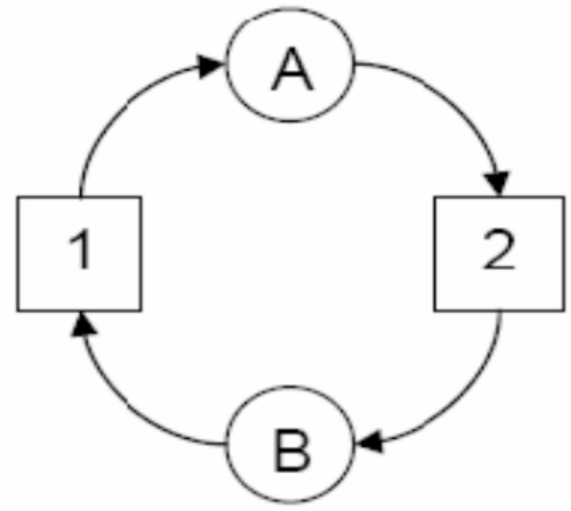
MONASH University

Introduction to Deadlock

- **One or more processes request resource(s) but never receive the resource(s).**
- **Deadlock referred to as a deadly embrace**
  - One task holds a resource requested by another task that holds a resource requested by the first task.
- **Resource allocation often managed with semaphores**

  Requesting a resource = waiting on the semaphore

  Releasing a resource = signalling the semaphore



MONASH University

Simple Example of Deadlock

TASK1:

```
wait(mutex_sem)

    critical section code 1 …

// whoops,
// forgot signal(mutex_sem )
```

TASK2:

```
wait(mutex_sem)

    critical section code 2 …

signal(mutex_sem)
```

MONASH University

Simple Example of Deadlock

TASK1:

```
wait(mutex_sem)
if (shared_var == 10)
{
    signal(mutex_sem);
    //do some work…
}
```

TASK2:

```
wait(mutex_sem)
    shared_var++
signal(mutex_sem)
```

MONASH University

Example 2: Deadly Embrace

```
TASK1:

wait(sem1)
    wait(sem2)

    …

    signal(sem2)
signal(sem1)
```

```
TASK2:
wait(sem2)


    wait(sem1)
        .....

    signal(sem1)
signal(sem2)
```

Example 2: Deadly Embrace

TASK1:

context switch

```
wait(sem1)
    wait(sem2)
    ...
    signal(sem2)
signal(sem1)
```

sem1 held by TASK1

TASK2:

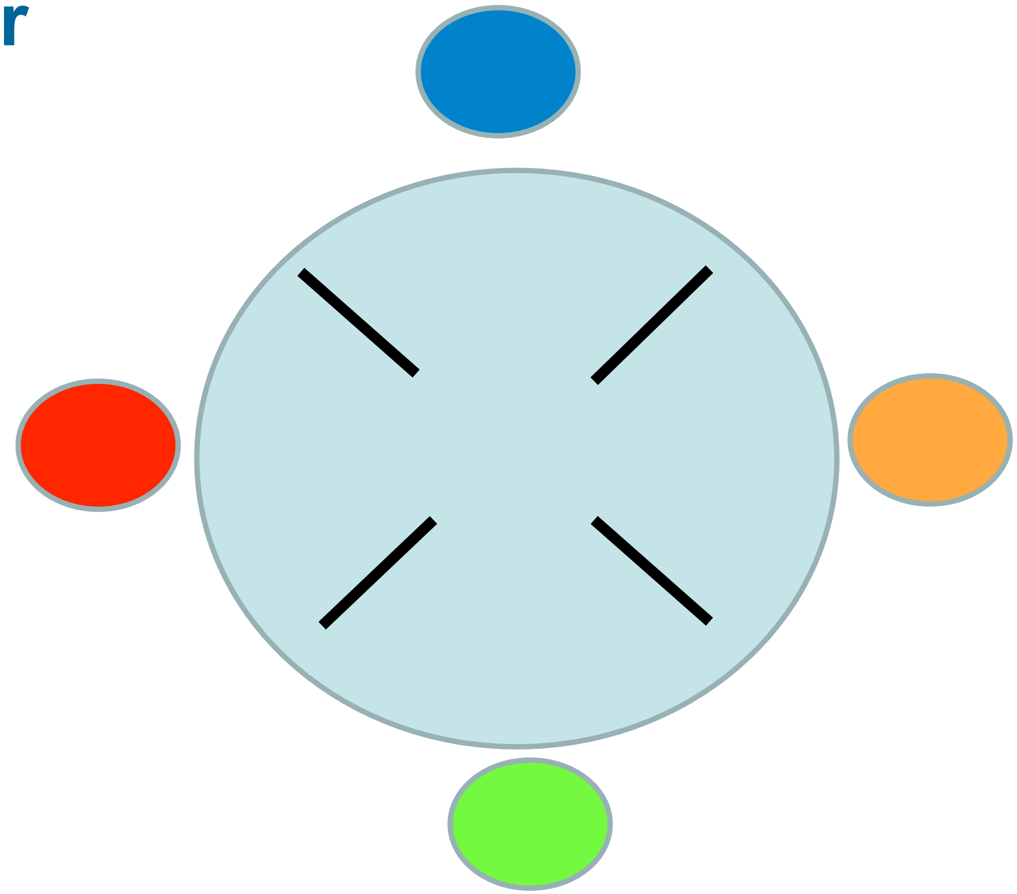
```
wait(sem2)
    wait(sem1)
    .....
    signal(sem1)
signal(sem2)
```

sem2 held by TASK2

MONASH University

Example 3: Philosophers

- **Philosophers either think or eat**
- **Philosophers need 2 chopsticks to eat**



Example 3: Philosophers

PhilGreen:

```
wait(sem1)
    …
    wait(sem2)

    …

    signal(sem2)
signal(sem1)
```

PhilBlue :

```
wait(sem2)
    …
    wait(sem3)

    …

    signal(sem3)
signal(sem2)
```

PhilYellow :

```
wait(sem3)
    …
    wait(sem4)

    …

    signal(sem4)
signal(sem3)
```

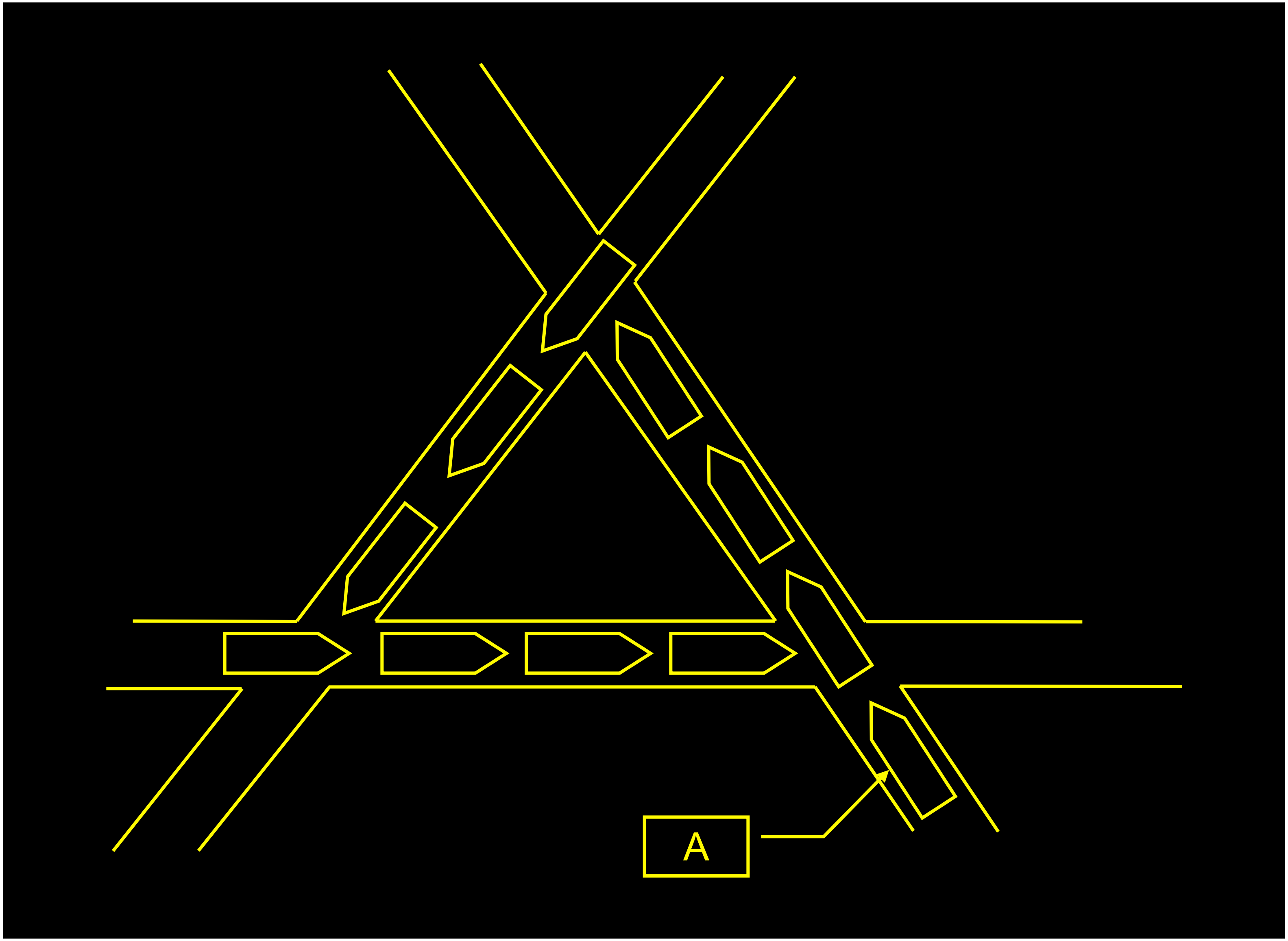
PhilRed:

```
wait(sem4)
    …
    wait(sem1)

    …

    signal(sem1)
signal(sem4)
```

Deadlock in City Traffic



Gridlock

- Each car holding a resource
  - position on the street
- Requesting another resource
  - move forward
- Being held by another
  - the car in front
- Car A is held up
  - but not part of the deadlock

Solution to Deadlock with Multiple resources

- **Acquire resources in the *same order* in all tasks.**
- **Document that ordering globally!**
  - eg sem1, sem2, cout, printer port, CS,

- **Wait on semaphores and mutexes in *same order* in all tasks**
  - eg wait(sem1); acquire printer; print, enter CS

    leave CS, release printer, signal(sem1);

What can we do about deadlock?

4 Basic Approaches:

Prevention

- Ensure 1 or more of the 3 necessary conditions for deadlock cannot occur.

Avoidance

- Let the OS know the maximum number of each resource it may need.

Detection

- Let deadlock happen and then recover

Preemption

- Suspend processes until resources are available

MONASH University

The Producer-Consumer Problem

There is a classic multiprocess synchronisation problem called either the Producer-consumer problem or the bounded-buffer problem.
It concerns two processes:
One is a producer of data,
One is a consumer of data.
Sharing a common fixed size buffer.

The Producer-Consumer Problem

The producer should not try to add data to the buffer if it's full.

The consumer should not try to remove data from an empty buffer.

We will look at a solution for one producer and one consumer but it works for multiple producers and consumers.

The Producer-Consumer Problem

The solution is to block the producer if the buffer is full. Each time the consumer removes an item from the buffer it signals the producer to start to fill the buffer again. In the same way the consumer blocks if it finds the buffer empty. Each time the producer puts data into the buffer it signals the consumer.

The Producer-Consumer Problem

The initialisation:

semaphore mutex = 1
semaphore full = 0
semaphore empty = buffer-size

The Producer-Consumer Problem

```
procedure producer() {
    while (true) {
        item =produceItem()
        wait(empty)
        wait(mutex)
        putItemIntoBuffer(item)
        signal(mutex)
        signal(full)
    }
}
```

```
procedure consumer() {
    while (true) {
        wait(full)
        wait(mutex)
        item = removeItemFromBuffer()
        signal(mutex)
        signal(empty)
        consumeItem(item)
    }
}
```

Deadlock in Consumer Producer Example

- **Refer to Example6_multibuffer.c on the unit webpage:**

In consumer task:

CS(cout<< “consumed” << retrieve_item());

In producer task:

CS(cout<< produced “<< i);

store_item();

Deadlock in Consumer Producer Example

- **Refer to Example6_multibuffer.c on the webpage:**

contains code:

wait_semaphore(full_places)

In consumer task:

CS(cout<< “consumed” << retrieve_item());

In producer task:

CS(cout<< produced “<< i);

store_item();

contains code:

signal_semaphore(full_places)

MONASH University

Deadlock in Consumer Producer Example

- **Refer to Example6_multibuffer.c:**

contains code:

In consumer task:

wait_semaphore(full_places)

CS(cout<< “consumed” << retrieve_item());

Stuck here waiting for full_places and holding CS

In producer task:

Stuck here waiting for CS and cannot signal full_places

CS(cout<< produced “<< i);
store_item();

contains code:

signal_semaphore(full_places)

MONASH University

Solution to Deadlock in Consumer Producer Example

In consumer task:

```
item = retrieve_item();
CS(cout<< “consumed” << item );
```

In producer task:

```
CS(cout<< produced “<< i);
store_item();
```

MONASH University

Lesson Learnt

- **Critical section CS macro should not contain code that blocks**

- **CS reserved for lowest level of synchronisation**